ソフトウェア実験 II・III
テーマ 2, 3　レポート

学籍番号：1TE22028G

氏名：一瀬遥希

提出日：　2025 年 1 月 23 日

テーマ 2：マルチタスクカーネルの製作

1. テーマ 2 についての説明

テーマ 2 では,C 言語で記述された複数のタスクを適切なタイミングで切り替えながら実行する「マルチタスクカーネル」を作成する.

タスクの切り替えは以下の 2 つの状況で発生する.

1. 共有資源を利用する際の排他制御
   同じメモリ領域や I/O デバイスを複数のタスクが利用する場合，データ競合を防ぐためにセマフォを使用する．この際，P・V システムコールを利用し，リソースのロック（P 命令）と解放（V 命令）を管理することで，安全なマルチタスク環境を構築する．また,Ready キュー（実行待ちタスクの管理）やセマフォキュー（特定の共有資源を待機しているタスクの管理）を活用し，タスクの状態を制御する.

2. CPU の独占防止のためのタイマ制御
   一定時間経過後にタイマ割り込みを発生させ，強制的にタスクを切り替える．これにより，複数のタスクが公平に CPU を利用できるようになる.

さらに，各タスクの状態を管理するために「タスクコントロールブロック（TCB)」を用いる．これにはプログラムカウンタ，レジスタ，スタックの内容などを保存・復元する機能が含まれる．また，カーネルがタスクの切り替え中に割り込みを受けないよう，割り込みマスクを用いたカーネル再入防止の仕組みも含める.

これらを実現するうえで，以下の基本事項への理解が重要である.

1. 特権命令の概念
   OS において，すべての命令が実行できるスーパバイザモードと一部の命令が実行できないユーザモードを理解し，スーパバイザモードでのみ実行できる特権命令を活用する．特に，入出力やタイマの設定などはこの特権命令として，必要なときのみスーパバイザモードに切り替えて処理を行うことで，システムに重大な障害を与えるのを防ぐ事ができる.

2. P・V システムコールの理解
   P・V 命令は，複数のタスクが共有資源を同時に操作することでデータの整合性が

崩れるのを防ぐための排他制御とタスク同期の手法である．セマフォはカウンタと待ち行列で構成され，P 命令は資源の取得（カウンタ減少），V 命令は資源の解放（カウンタ増加）を行う．カウンタが負ならタスクを待機，0 なら継続，正なら何もしない．P・V 命令により，資源のロック管理やタスク間の処理順制御が可能となる．これらは OS の特権命令としてシステムコールで実装される．

3. タスク切り替えの仕組み

   マルチタスク処理では，タスクの途中で実行が中断されても，再開後の結果が切り替えなしの場合と同じになる必要がある．そのため，中断時に計算機の状態を正しく保存し，再開時にその状態を回復することが重要となる．具体的には，中断時のプログラムカウンタの値，レジスタの値，スタックの内容を保存し，再開時に復元することで，タスクの継続実行を可能にする．

4. カーネル再入防止

   タスクの切替えは，P 命令による休眠状態への移行とタイマ割込みによって行われる．P 命令はユーザタスクのみが発行するため，カーネルタスクの実行中には発生しないが，タイマ割込みはカーネルタスクの実行中にも発生する可能性があり，不適切なタイミングでの切替えにより管理情報が破壊される危険がある．この問題を防ぐため，カーネルタスクの実行中に発生したタイマ割込みではタスク切替えを行わず，割込み前の状態に復帰する方法が一般的（UNIX の方式）だが，今回の実験では，カーネルタスクの実行中はタイマ割込み自体を受け付けないことで対処する．具体的には，P・V 命令処理のアセンブラルーチンの最初で割込みを禁止（走行レベル 7）し，最後で元に戻すことで実現する．ただし，この方法ではタスクの連続実行時間の上限が厳密に一定とはならず，システムコール実行中はタスク切替えが遅れる可能性がある．

2. テーマ 2 の実験全体における位置づけ

テーマ 2 においては，マルチタスクカーネルの製作を通じて，タスクの並列実行をシミュレートし，OS が行うタスクスケジューリングの基本概念や共有資源に対する排他制御などを学ぶことに重点が置かれている．このように，テーマ 2 は OS の基本機能の習得を目的とした基礎フェーズであり，テーマ 3 において 3 つ以上のタスクを扱ったり，2 つのポートを使ってマルチタスキングを行えるように機能を拡張していくにあたっての重要な準備段階に位置づけられると考えられる．

3. プログラムのリスト

以下に，テーマ 2 において作成を行ったプログラムファイル mtk_c.c, mtk_c.h, mtk_asm.s, test2.c のリストを示す.

・mtk_c.c

```
#include <stdio.h>
#include "mtk_c.h"

/*=========================================================
変数および配列の定義
=========================================================*/
SEMAPHORE_TYPE semaphore[NUMSEMAPHORE]; /* セマフォ */
TCB_TYPE task_tab[NUMTASK + 1];         /* TCB 配列 */
STACK_TYPE stacks[NUMTASK];             /* タスクスタック */

/* 大域変数 */
TASK_ID_TYPE curr_task;
TASK_ID_TYPE new_task;
TASK_ID_TYPE next_task;
TASK_ID_TYPE ready;

/*=========================================================
カーネルの初期化
init_kernel:
引数   : なし
返り値 : なし
担当   : 若松
=========================================================*/

void init_kernel()
{
  /* TCB 配列の初期化．TCB 配列の各要素は TCB_TYPE 型の構造体 */
  for (int i = 0; i <= NUMTASK; i++)
  {                                 /* id = 0 は使わないが初期化は行っておく */
    task_tab[i].status = EMPTY;     /* status = EMPTY */
    task_tab[i].next = NULLTASKID; /* 次のタスク id は NULLTASKID */
    task_tab[i].task_addr = NULL;  /* タスクアドレス = NULL */
```

```
    task_tab[i].stack_ptr = NULL;  /* スタックアドレス = NULL*/
    task_tab[i].priority = 0;      /* 優先度は 0 */
  }

  /* ready キューの初期化 */
  ready = 0; /* 実行待ちタスクはない */

  /* pr_handler = TRAP #1 */
  *(unsigned int *)0x084 = (unsigned int)pv_handler; /* 関数名でアドレス参照 */

  /* セマフォの値を初期化 */
  for (int i = 0; i < NUMSEMAPHORE; i++)
  {
    semaphore[i].count = 1;              /* セマフォは専有されていない */
    semaphore[i].task_list = NULLTASKID; /* 初期化時点で次のタスクはない */
  }
}

/*=========================================================
ユーザタスクの初期化と登録
set_task:
引数 : ユーザタスク関数へのポインタ(タスクの先頭番地)
返り値 : なし
担当:一瀬
=========================================================*/

void set_task(void *p)
{
  TASK_ID_TYPE i;
  for (int i = 1; i <= NUMTASK; i++)
  {
    if (task_tab[i].status == EMPTY)
    {
      new_task = i;
      break;
    } // タスクを走査し、空きスロット見つけたらその ID を new_task に代入
```

```
  }
  task_tab[new_task].task_addr = p;                    // 空きスロットにユーザタスク関数のポ
インタを代入
  task_tab[new_task].status = OCCUPIED;                // スロットの使用状態を使用中に変更
  task_tab[new_task].stack_ptr = init_stack(new_task); // init_stack で初期化
  addq(&ready, new_task);                              // ready キューに new_task を登録
}

/*==========================================================================
ユーザタスク用のスタックの初期化
init_stack:
引数   : タスク ID
返り値 : 初期化後にユーザタスク用 SSP が指すアドレス(void *型)
担当:若松、一瀬(共同)
==========================================================================*/

void *init_stack(TASK_ID_TYPE id)
{
  int *int_ssp;
  int_ssp = (int *)&stacks[id - 1].sstack[STKSIZE]; // S スタックの底+1 のアドレスをポインタに
代入

  *(--int_ssp) = (int)task_tab[id].task_addr; // 初期 PC を積む

  unsigned short int *short_ssp = (unsigned short int *)int_ssp;
  *(--short_ssp) = 0x0000; // 初期 SR を積む
  int_ssp = (int *)short_ssp;

  int_ssp -= 15;                                  // 15×4 バイト分の領域を飛ばす(レジスタ群を
積む)
  *(--int_ssp) = &stacks[id - 1].ustack[STKSIZE]; // 初期 USP を積む

  return (void *)int_ssp; // 関数終了時の SSP が返り値
}

/*==========================================================================
```

```
マルチタスク処理の開始
begin_sch:
引数 : なし
返り値 : なし
担当 : 若松
================================================================================*/

void begin_sch()
{
  curr_task = removeq(&ready); /* removeq で最初のタスクを設定 */
  init_timer();                /* init_timer でタイマの初期化 */
  first_task();                /* first_task で最初のタスクを起動 */
}

/*================================================================
タスクキューの最後尾への TCB の追加
addq:
引数 : キューへのポインタとタスクの ID
返り値 : なし
担当:一瀬
================================================================*/

void addq(TASK_ID_TYPE *que_ptr, TASK_ID_TYPE id)
{
  if (*que_ptr == NULLTASKID)
  {               // キューの先頭のタスクが空なら,
    *que_ptr = id; // 先頭にタスクを登録
  }
  else
  {
    TCB_TYPE *task_ptr = &task_tab[*que_ptr]; // 先頭のタスクのポインタ設定
    while (1)
    {
      if ((*task_ptr).next == NULLTASKID)
      {                       // その次のタスクが空だったら,
        (*task_ptr).next = id; // タスクを登録
```

```
        break;
      }
      else
      {
        task_ptr = &task_tab[(*task_ptr).next]; // 次のタスクにポインタを移動
      }
    }
  }
}

/*=============================================================
タスクキューの先頭から TCB の除去
removeq:
引数 : キューへのポインタ
返り値 : 先頭の ID
担当:一瀬
=============================================================*/
TASK_ID_TYPE removeq(TASK_ID_TYPE *que_ptr)
{
  TASK_ID_TYPE r_id = *que_ptr; // 返却値(先頭のタスクの id を取得)
  if (r_id != NULLTASKID)
  {                                       // キューの先頭が空でなければ
    TCB_TYPE *task_ptr = &task_tab[r_id]; // 先頭のタスクのポインタ設定
    *que_ptr = (*task_ptr).next;          // 先頭から 2 番目のタスクを先頭にする
    (*task_ptr).next = NULLTASKID;        // 先頭のタスクの next は NULLTASKID にして、タスクを
取り出す
  }
  return r_id; // キューの先頭のタスクの id を返す
}

/*=============================================================
タスクのスケジュール関数
sched:
引数 : なし
返り値 : なし
担当 : 若松
```

```
================================================================*/

void sched()
{
  next_task = removeq(&ready); /* ready キューからタスクを取り出し next_task に代入 */
  while (next_task == NULLTASKID)
  {
    while (1)
      ;
  } /* NULLTASKID の場合は無限ループ */
}

void p_body(int id)
{
  semaphore[id].count--; /*セマフォの値をデクリメント*/
  if (semaphore[id].count < 0)
  {
    /*その結果セマフォが獲得できないなら*/
    sleep(id); /*セマフォ待ちのキューに入れ休眠状態へ*/
  }
}

void v_body(int id)
{
  semaphore[id].count++; /*セマフォの値をインクリメント*/
  if (semaphore[id].count <= 0)
  {             /*その結果セマフォが空いたなら*/
    wakeup(id); /*そのセマフォを待っている先頭のタスクを実行可能状態へ*/
  }
}

int sleep(int a)
{ /* a := セマフォ ID*/
  addq(&(semaphore[a].task_list), curr_task);
  sched();
  swtch();
```

```
}

int wakeup(int a)
{
  TASK_ID_TYPE task_id = removeq(&(semaphore[a].task_list));
  addq(&ready, task_id);
}
```

・**mtk_c.h**

```
#ifndef MTK_C_H
#define MTK_C_H

// カーネル関連の定数定義//////////////////////////////////////////
#define NUMSEMAPHORE 5
#define NULLTASKID 0
#define NUMTASK 5
#define STKSIZE 1024

// task_tab.status 定数/////////////////////////////////////////////
#define EMPTY 0
#define OCCUPIED 1
#define COMPLETED 2

// 構造体の定義/////////////////////////////////////////////////////

typedef int TASK_ID_TYPE; /* TASK_ID_TYPE の定義 */

typedef struct
{
  int count;
  int nst; /* reserved */
  TASK_ID_TYPE task_list;
} SEMAPHORE_TYPE;

typedef struct
{
```

```
  void (*task_addr)();
  void *stack_ptr;
  int priority;
  int status;
  TASK_ID_TYPE next;
} TCB_TYPE;

typedef struct
{
  char ustack[STKSIZE];
  char sstack[STKSIZE];
} STACK_TYPE;

// 大域変数、配列の定義////////////////////////////////////////

extern TASK_ID_TYPE curr_task;
extern TASK_ID_TYPE new_task;
extern TASK_ID_TYPE next_task;
extern TASK_ID_TYPE ready;

extern SEMAPHORE_TYPE semaphore[NUMSEMAPHORE];
extern TCB_TYPE task_tab[NUMTASK + 1];
extern STACK_TYPE stacks[NUMTASK];

// 関数のプロトタイプ宣言////////////////////////////////////////
extern void outbyte(int ch, unsigned char c);
extern char inbyte(int ch);

/// マルチタスクカーネル関連の関数

extern void init_kernel();
extern void set_task(void *p);
extern void *init_stack(TASK_ID_TYPE id);
extern void begin_sch();
extern void addq(TASK_ID_TYPE *que_ptr, TASK_ID_TYPE id);
extern TASK_ID_TYPE removeq(TASK_ID_TYPE *que_ptr);
```

```
extern void sched();

extern void first_task();
extern void swtch();

/// タイマ関連の関数
extern void hard_clock();
extern void init_timer();
///// モニタ内の関数
extern void set_timer();
extern void reset_timer();

/// セマフォ関連の関数
extern void p_body(int a);
extern void v_body(int a);

extern int sleep(int a);
extern int wakeup(int a);

extern void P(int id);
extern void V(int id);
extern void pv_handler();

extern void skipmt();

#endif
```

・**mtk_asm.s**

```
**global 変数の宣言
.global first_task
.global swtch
.global hard_clock
.global init_timer
.global skipmt
.global P
```

```
.global V
.global pv_handler

**外部入力(大域変数)
.extern curr_task
.extern next_task
.extern ready
.extern task_tab

**外部関数
.extern addq
.extern sched
.extern p_body
.extern v_body

**システムコール番号
.equ SYSCALL_NUM_GETSTRING, 1
.equ SYSCALL_NUM_PUTSTRING, 2
.equ SYSCALL_NUM_RESET_TIMER,   3
.equ SYSCALL_NUM_SET_TIMER, 4
.equ SYSCALL_NUM_SKIPMT, 5

.equ PV_CALL_P,         0
.equ PV_CALL_V,         1

.section .text

*****************************************************************************
** ユーザタスク起動用ルーチン
** first_task
** 入出力なし
** 担当:一瀬
**
*****************************************************************************
first_task:
    move.l  curr_task, %d0            /*curr_task の番号を d1 に*/
```

```
    movea.l #task_tab, %a0              /*task_tab 配列の先頭アドレスを a0 に*/

    mulu.w  #20, %d0                   /*curr_task の番号に 20 を乗算し、d1 に格納*/
    add.l   %d0, %a0                    /*a0 に d1 を加算し、curr_task の先頭アドレスを計算*/

    movea.l %a0, %a1
    add.l   #4,  %a1                    /*a0 は該当の curr_task の stack_ptr(SSP)の先頭アドレ
ス*/

    move.l  (%a1), %sp                  /*スーパーバイザーモードの sp に SSP を回復*/
    move.l  (%sp)+, %a2
    move.l  %a2, %USP

    movem.l (%sp)+, %d0-%d7/%a0-%a6     /*レジスタ 15 本回復*/

    rte

*****************************************************************************
** タスクスイッチを実際に起こす関数
** swtch
** 入出力なし
** 担当:若松
*****************************************************************************
swtch:
    move.w  %SR,-(%sp)                  /* SR を退避し rte での復帰を可能に */

    movem.l %d0-%d7/%a0-%a6,-(%sp)      /* 実行中タスクのレジスタ退避 */
    move.l  %usp, %a0
    move.l  %a0, -(%sp)

    move.l  curr_task, %d1              /* curr_task -> d1 */
    mulu.w  #20, %d1                    /* TCB 配列の各要素は 4*5=20byte，タスク id に乗算 */
    movea.l #task_tab, %a0              /* task_tab 配列の先頭アドレス -> a0 */
    add.l   %d1, %a0                    /* task_tab 配列内の curr_task の先頭アドレスまで移動
*/
    add.l   #4,  %a0                    /* stack_ptr(SSP)の先頭アドレス -> a0 */
```

```
    move.l  %sp, (%a0)                  /* 現在のタスクの TCB に SSP を記録*/

    move.l  next_task, %d1              /* next_task-> d1 */
    move.l  %d1, curr_task              /* curr_task を next_task で更新*/
    mulu.w  #20, %d1                    /* TCB 配列の各要素は 4*5=20byte，タスク id に乗算 */
    movea.l #task_tab, %a0              /* task_tab 配列の先頭アドレス -> a0*/
    add.l   %d1, %a0                    /* task_tab 配列内の curr_task の先頭アドレスまで移動
*/
    add.l   #4,  %a0                    /* stack_ptr(SSP)の先頭アドレス -> a0 */
    move.l  (%a0),%sp                   /* スーパバイザモードの sp に SSP を回復 */

    move.l  (%sp)+, %a0
    move.l  %a0, %usp
    movem.l (%sp)+,%d0-%d7/%a0-%a6      /*次のタスクのレジスタを回復*/

    rte

hard_clock:
    movem.l %d0-%d7/%a0-%a6, -(%sp)

    move.l  #ready, %d0          |%d0 -> ready キューへのポインタ
    move.l  curr_task, %d1           |%d1 -> タスクの ID
    movem.l %d0-%d1, -(%sp)      |%d0，%d1 をスタックに積んで
    jsr addq                 |addq を実行
    adda.l  #8, %sp          |%d0，%d1 の 8 バイト分を%sp に加算

    jsr sched
    jsr swtch

    movem.l (%sp)+, %d0-%d7/%a0-%a6

    rts

init_timer:
    movem.l %d0-%d2, -(%sp)
    move.l #SYSCALL_NUM_RESET_TIMER, %d0     | タイマーをリセット
```

```
    trap #0

    move.l #SYSCALL_NUM_SET_TIMER, %d0      | タイマーをセット
    move.w #10000, %d1                                          | およそ1秒くらい
    move.l #hard_clock, %d2                                     | 割り込み時に呼び出す
ルーチンをセット
    trap #0

    movem.l (%sp)+,%d0-%d2
    rts

skipmt:
    movem.l %d0-%d2, -(%sp)
    move.l #SYSCALL_NUM_SKIPMT, %d0
    trap    #0

    movem.l (%sp)+,%d0-%d2
    rts

P:
    link.w  %a6,#0
    movem.l %d0-%d1,-(%sp)  /*レジスタ退避*/

    move.l  #0,%d0          /*P システムコール ID の 0 を d0 レジスタにセット*/
    move.l  8(%a6),%d1      /*スタックから取り出した引数(セマフォ ID)を d1 レジスタにセット*/
    trap    #1

    movem.l (%sp)+,%d0-%d1  /*レジスタ復帰*/
    unlk    %a6

    rts

V:
    link.w  %a6,#0
    movem.l %d0-%d1,-(%sp)  /*レジスタ退避*/
```

```
    move.l  #1,%d0          /*V システムコール ID の 1 を d1 レジスタにセット*/
    move.l  8(%a6),%d1      /*スタックから取り出した引数(セマフォ ID)を d1 レジスタにセット*/
    trap    #1

    movem.l (%sp)+,%d0-%d1  /*レジスタ復帰*/
    unlk    %a6

    rts

pv_handler:
    movem.l %a0-%a6/%d0-%d7, -(%sp)
    move.w  %SR,-(%sp)  /*SR をスタックに退避*/
    move.w  #0x2700,%SR /*走行レベルを7にして割り込み禁止*/

    movem.l %d1,-(%sp)  /*レジスタ d1 をスタックに退避*/
    cmpi.l  #0,%d0      /*d0 の値が 0 であるか比較*/
    bne pv_handler_1    /*d0 が 0 でないならば分岐*/
    jsr p_body          /*p_body()を呼び出す*/
    bra pv_handler_end  /*復帰処理へ*/

pv_handler_1:
    cmpi.l  #1,%d0      /*d0 の値が 1 であるか比較*/
    bne pv_handler_end  /*d0 が 1 でないならば分岐*/
    jsr v_body          /*v_body()を呼び出す*/

pv_handler_end:

    movem.l (%sp)+,%d1  /*レジスタ d1 をスタックから復帰*/
    move.w  (%sp)+,%SR  /*SR をスタックから復帰*/
    movem.l (%sp)+, %a0-%a6/%d0-%d7
    rte                 /*割り込み終了*/
```

・**test2.c**

```
#include <stdio.h>
#include "mtk_c.h" // マルチタスクカーネル用ヘッダー
int N=3;
```

```
int K=10000;
volatile int nttask;

int task_0(){
    while(1){
        printf(" task0 is runnnig¥n");
        P(0);
        if(nttask == N){
            nttask=0;
            for (int i=0; i<N; i++){
                printf("  do V(1) ,%d times¥n",i+1);
                V(1);
            }
            printf("tasks reset¥n");
        }
        V(0);
        printf("jmp to skipmt¥n");
        skipmt();
        printf("task0 is back¥n");
    }
}

int task1(){
    while(1){
        printf(" task1 start¥n");
        for(int k=0; k<K; k++){
            if(k%1000==0){
                printf("task1 ,%d times¥n",k/1000);
            }
        }
        printf(" task1 is runnnig¥n");
        P(0);
        nttask++;
        printf("nttask is %d now¥n",nttask);
        V(0);
        printf("             task1 is finished¥n");
```

```
        P(1);
    }
}

int task2(){
    while(1){
        printf(" task2 start¥n");
        for(int k=0; k<2*K; k++){
            if(k%1000==0){
                printf("task2 ,%d times¥n",k/1000);
            }
        }
        printf(" task2 is runnnig¥n");
        P(0);
        nttask++;
        printf("nttask is %d now¥n",nttask);
        V(0);
        printf("              task2 is finished¥n");
        P(1);
    }
}

int task3(){
    while(1){
        printf(" task3 start¥n");
        for(int k=0; k<3*K; k++){
            if(k%1000==0){
                printf("task3 ,%d times¥n",k/1000);
            }
        }
        printf(" task3 is runnnig¥n");
        P(0);
        nttask++;
        printf("nttask is %d now¥n",nttask);
        V(0);
        printf("             task3 is finished¥n");
```

```
        P(1);
    }
}

int main() {

    printf("Initializing kernel...¥n");
    init_kernel();
    printf("Kernel initialized.¥n");

    semaphore[0].count =1;
    semaphore[1].count =0;
    nttask = 0;

    printf("Setting Task 0...¥n");
    set_task((char *)task_0); // タスクを登録
    printf("Task 0 set.¥n");

    printf("Setting Task 1...¥n");
    set_task((char *)task1); // タスクを登録
    printf("Task 1 set.¥n");

    printf("Setting Task 2...¥n");
    set_task((char *)task2); // タスクを登録
    printf("Task 2 set.¥n");

    printf("Setting Task 3...¥n");
    set_task((char *)task3); // タスクを登録
    printf("Task 3 set.¥n");

    printf("Starting scheduler...¥n");
    begin_sch();

    // begin_sch 後のコードは実行されないはず
    printf("Scheduler started (this line should not be printed).¥n");
```

```
    return 0;
}

/* exit() defined in test*.c */
void exit(int value) {
    *(char *)0x00d00039 = 'h';  /* led0 への表示 (halt) */
    for (;;)
    ;               /* 無限ループトラップで停止させる */
}
```

4. プログラムの説明

以下に，上記に示したプログラムファイル毎に，プログラムの説明を示す．

mtk_c.c

・カーネルの初期化：init_kernel();

この関数では，引数を取らず，TCB 配列・ready キュー・セマフォの値の初期化を行い，pv_handler を TRAP #1 の割り込みベクタに登録する．なお，TCB 配列の各要素は TCB_TYPE 型の構造体である．ここでは，後々スケジューリングなどに使用することも踏まえて，TCB の使用状態を記号定数としてヘッダファイルに宣言し，可読性の工場に努めている．

・ユーザタスクの初期化と登録：set_task();　担当部

引数には，ユーザタスク関数へのポインタ（タスク関数の先頭番地）void *p を取る．まず，task_tab[]を走査して空きスロットを見つけ，その ID を new_task に代入する．つぎに，上で見つけた TCB の更新を行い，スロットを使用中に変更する．その後，init_stack(); の値を，その TCB の stack_ptr に登録し，最後に addq(); を用いて，ready キューに new_task を登録する．

・ユーザタスク用のスタックの初期化：init_stack(); 担当部

引数には，タスク ID を取り，初期化後のスーパバイザスタックポインタ（SSP）が指すアドレスを void* 型で返す。以下に処理の詳細な流れを箇条書きで示す．

1. スタックのアドレスを設定
   タスク id に対応する stacks[id - 1].sstack のスタック底（+1 の位置）をポインタ int_ssp に格納する．

2. 初期プログラムカウンタ（PC）を積む
   task_tab[id].task_addr（タスクの開始アドレス）をスタックにプッシュし，初期 PC として設定する．

3. 初期ステータスレジスタ（SR）の設定
   0x0000 をプッシュし，SR（ステータスレジスタ）を初期化する．

4. レジスタ領域の確保
   レジスタ保存用に 15×4 バイト分スタックを確保（スタックポインタを進める）．

5. 初期ユーザスタックポインタ（USP）の設定
   stacks[id - 1].ustack[STKSIZE]（ユーザスタックのトップ）をプッシュし，USP（ユーザタスク用スタックポインタ）を設定する．

6. スーパバイザスタックポインタ（SSP）の返却
   初期化完了時の int_ssp を void* 型で返す．

処理の中で，*(--ssp) = 値の形式を用いることで，スタックに値をプッシュする際に，スタックポインタをデクリメントしながら代入する動作を簡単に実装している．また，int 型ポインタと unsigned short int 型ポインタを適切に使い分けることで，4 バイト（整数）や 2 バイト（ステータスレジスタ）といった異なるデータサイズを正しく格納できるようにしている．

・マルチタスク処理の開始 begin_sch();

この関数は，引数を取らない．まず，removeq()関数で ready キューから最初のタスクを取り出し，curr_task に代入する．次に，init_timer(); でタイマを設定する．そして，first_task(); で最初のタスクを機動する．

・タスクキューの最後尾への TCB の追加：addq(); 担当部

引数として，キューへのポインタとタスクの ID を受け取る．実際に作成した処理の流れ

の詳細を以下に箇条書きで示す.

1. キューが空の場合（*que_ptr == NULLTASKID）
   先頭のタスクが存在しないため，キューの先頭に id を登録する.

2. キューにタスクがすでに存在する場合
   task_tab[*que_ptr] でキューの先頭タスクの TCB（タスク制御ブロック）**を取得し，最後尾のタスクを探すために,next が NULLTASKID (次のタスクがない) までループを行う. 最後尾のタスクが見つかったら，その next に id を登録し，タスクをキューの最後尾に追加する.

この関数は，タスクキューの管理を行い，新しいタスクを適切に最後尾へ追加する役割を持つ. キューが空の場合は，新しいタスクを先頭に直接登録し，すでにタスクが存在する場合は，最後尾の next を更新して新しいタスクを接続する. while (1)ループを用いてキューの末尾を探索し，ポインタを適切に更新しながら処理を進めることで，リストの順序を維持する仕組みとなっている. これにより，タスクの実行順を保ちながら，新しいタスクをスムーズにキューへ追加できるようになっている.

・タスクのスケジュール関数：sched();

この関数は，引数を取らない. removeq(); で ready キューの先頭のタスク ID を取り出し，next_task に代入する. ただし, ready キューが空の場合は無限ループに入る.

・P システムコールの本体：p_body();

引数には，セマフォ ID を取る. セマフォの値をデクリメントし，セマフォが獲得できなければ, sleep(セマフォ ID)を実行し，休眠状態に入る.

・V システムコールの本体：v_body();

引数には，セマフォ ID を取る. セマフォの値をインクリメントし，セマフォが空けば，wakeup(セマフォ ID)を実行し，そのセマフォを持っているタスクを実行可能状態にする.

・タスクを休眠状態にしてタスクスイッチをする：sleep(int a);

引数には，チャンネル ch を取る. (ここでは，作成の都合上簡単のために int a としてい

る．）スケジューラ sched(); を起動して，次に実行するタスクのタスク ID を next_task にセットする．その後，タスクスイッチ関数である swtch を呼び出し，タスクの切り替えを行う．

・休眠状態のタスクを実行可能状態にする：wakeup(int a);

引数には，チャンネル ch を取る．（ここでは，作成の都合上簡単のために int a としている．）まず，removeq(); でチャンネル ch の待ち行列に繋がっている先頭のタスクを取り除き，その ID を TASK_ID_TYPE task_id に代入する．次に，先程の task_id を利用して，addq(); でそのタスクを ready キューに繋ぎ直して，そのタスクを実行可能状態にする．

mtk_c.h

mtk_c.h は，マルチタスクカーネルの構築に必要なタスク管理，スケジューリング，セマフォ制御，タイマ管理のための構造体，変数，関数を定義した重要なヘッダファイルである．このヘッダファイルを使用することで，C 言語ベースのマルチタスク環境を効果的に実装できる．以下に，このヘッダファイル内で定義されているものの説明を示す．

・カーネル関連の定数の定義

NUMSEMAPHORE：使用可能なセマフォの最大数
NULLTASKID：タスクキューが空であることを示す値
NUMTASK：許容できる最大タスク数
STKSIZE：タスクスタックのサイズ(1KB としている)

・task_tab.status 定数

EMPTY：タスクスロットが未使用
OCCUPIED：タスクスロットが使用中
COMPLETED：タスクが終了した状態

・構造体の定義

ここでは，マルチタスクの制御に必要なデータを管理するため構造体が定義されている．

TASK_ID_TYPE：
タスク ID を int 型で定義している．

SEMAPHORE_TYPE：
count：セマフォのカウント(リソースの使用可能数の管理用)．

nst：予約領域(今回は未使用)
task_list：セマフォ待機キューの先頭タスク ID (セマフォを待つタスクの管理用).

TCB_TYPE：
task_addr：タスクが実行する関数のアドレス(エントリーポイント)
*stack_ptr：タスクのスタックポインタ
priority：タスクの優先度
status：タスクの使用状態を表す（EMPTY, OCCUPIED, COMPLETED）
next：次のタスク ID (キューとして管理するためのポインタ)

STACK_TYPE：
ustack：ユーザモード用のスタック領域
sstack：スーパバイザモード用のスタック領域

・大域変数，配列の定義
ここでは，カーネルの状態やタスク情報を管理するグローバル変数が定義されている.

curr_task：現在実行中のタスクを示す
new_task：新しく登録されるタスク ID
next_task：次に実行するタスク ID
ready：実行可能なタスクのキューの先頭 ID
semaphore：セマフォ情報を保持する配列（最大 NUMSEMAPHORE 個）
task_tab：タスクの管理情報を格納する TCB テーブル（NUMTASK + 1）
stacks：タスクのスタック領域

・関数のプロトタイプ宣言
カーネルの基本機能を提供するための関数が定義されている.

mtk_asm.s
このプログラムファイルには，マルチタスクカーネルのうちアセンブリ言語で構成される関数を記述している.

・ユーザタスク起動サブルーチン：first_task 担当部

このルーチンは，カーネルが使用しているスタックを現在のタスク（curr_task）のスタックに切り替え，マルチタスク処理を開始するためのサブルーチンである．また，このルー

チンはスーパバイザモードで実行される必要がある．以下に，詳細な処理の流れを示す．

1. curr_task の TCB（タスクコントロールブロック）アドレスを取得
   curr_task は現在のタスク ID（整数値）であり，これをタスクテーブル task_tab の適切なインデックスに変換します．task_tab の各エントリ（TCB）のサイズは 20 バイトであるため，curr_task の番号を 20 倍して適切な TCB アドレスを求めている．

2. SSP（スーパバイザスタックポインタ）の回復
   curr_task の TCB の stack_ptr に保存されていた SSP を SP（スーパバイザスタックポインタ）に復元する．これにより，現在のタスクが実行を再開できる状態になる．

3. USP（ユーザスタックポインタ）の回復
   スーパバイザスタックに保存されていた USP（ユーザスタックポインタ）を復元する．これにより，タスクがユーザモードで実行するときに使用するスタックが正しく設定される．

4. 残りのレジスタの回復
   movem.l 命令を使用して，スーパバイザスタックから d0-d7 と a0-a6 を復元する．これにより，タスクが中断された時点の状態を完全に再現できる．

5. ユーザタスクの起動
   RTE（Return from Exception）命令を実行することで，カーネルモードからユーザタスクの実行を開始する．RTE は，保存されたプログラムカウンタ（PC）とステータスレジスタ（SR）を復元し，タスクの実行を再開する．

・タスクスイッチ関数：swtch

このルーチンは，現在のタスクを中断し、次に実行するタスクへ切り替える処理を行う関数です。sleep() やタイマ割り込み（hard_clock()）によって呼び出され、タスクのレジスタやスタックの状態を保存・復元することで、タスクの切り替えを実現します。以下に，詳細な処理の流れを示す．

1. SR（ステータスレジスタ）を保存
   RTE による復帰を可能にするため，現在の SR をスタックに積む．

2. 現在のタスクのレジスタを保存

d0-d7, a0-a6, USP をスーパバイザスタックに退避する.

3. 現在のタスクの SSP を TCB に保存
   curr_task の stack_ptr に現在の SSP を記録する.

4. curr_task を next_task に更新
   next_task に設定されている次のタスク ID を curr_task に代入する.

5. 次のタスクの SSP を復元
   新しい curr_task の stack_ptr から SSP を回復する.

6. 次のタスクのレジスタを復元
   USP, d0-d7, a0-a6 をスーパバイザスタックから復元する.

7. タスクの切り替えを実行
   RTE により, 次のタスクの実行を開始する.

・タイマ割り込みルーチン：hard_clock

このルーチンは, タイマ割り込みを行うルーチンである. まず, レジスタの対比を行い, addq(); で現タスクを ready キューの末尾に追加する. そして, sched(); および swtch を起動する.

・クロック割り込みルーチン：init_timer

このルーチンは, タイマをセットし、一定周期ごとにハードウェア割り込みを発生させるためのルーチンである. まず d0-d2 レジスタをスタックに保存し、タイマ設定の影響を受けないようにする。その後、SYSCALL_NUM_RESET_TIMER を d0 に設定し、trap #0 を実行してタイマをリセットする。続いて、SYSCALL_NUM_SET_TIMER を d0 に設定し、d1 = 10000（約 1 秒の割り込み周期）と d2 = hard_clock（割り込み時の処理ルーチン）を指定して、trap #0 によりタイマを有効化する。最後に d0-d2 のレジスタを復元し、rts で関数を終了する。

・低難度 Option：skipmt 担当部

このルーチンは, システムコール番号 5（SYSCALL_NUM_SKIPMT）を使用し, trap #0

を実行することで特定のマルチタスク処理をスキップするルーチンである.

・P システムコールの入口：P

このルーチンは, C 言語からセマフォ ID を引数として呼び出され, P 操作（セマフォ取得）をシステムコール trap #1 を通じて実行するためのルーチンである. まず a6 をスタックフレームとして確保し, d0-d1 のレジスタをスタックに保存する. 次に, d0 に P システムコールの ID（0）をセットし, スタックから取得したセマフォ ID を d1 に格納する. 次に, trap #1 を実行してカーネルの P システムコール(p_body())を呼び出し, セマフォの状態に応じてタスクの実行を継続または休眠させる. 処理終了後, d0-d1 のレジスタを復元し, a6 のスタックフレームを解除した後, rts により関数を終了し, C 言語の呼び出し元へ戻る.

・V システムコールの入口：V

このルーチンは, C 言語からセマフォ ID を引数として呼び出され, V 操作（セマフォ解放）をシステムコール trap #1 を通して実行するためのルーチンである. タスクが使用していたセマフォを解放し, 待機中のタスクがあれば実行可能状態にする. まず a6 をスタックフレームとして確保し, d0-d1 のレジスタをスタックに保存する. 次に. d0 に V システムコールの ID（1）をセットし, スタックから取得したセマフォ ID を d1 に格納する. trap #1 を実行してカーネルの V システムコール(v_body())を呼び出し, セマフォのカウントを増加させ, 待機中のタスクがあれば実行可能状態にする. 処理終了後, d0-d1 のレジスタを復元し, a6 のスタックフレームを解除した後, rts により関数を終了し, C 言語の呼び出し元へ戻る.

・TRAP #1 割り込み処理ルーチン：pv_handler

このルーチンは, P または V のシステムコール(trap #1)が実行されたときに割り込み処理を行うルーチンであり, スーパバイザモードで実行される. C 言語の pv_handler()として直接呼び出してはならず, P または V の TRAP #1 の実行により動作する.

D0 レジスタには P または V のシステムコールの種類が, D1 レジスタにはセマフォ ID がセットされた状態で呼び出される. このルーチンは, D0 の値に応じて p_body(ID)または v_body(ID)を呼び出し, セマフォ操作を実行する. 詳細な処理の流れを以下に示す.

1. レジスタを保存

割り込み処理の影響を受けないように，アドレスレジスタ(a0-a6)とデータレジスタ(d0-d7)をスーパバイザスタックに退避する．これにより，処理が終了した後に元のタスクの状態を正しく復元できるようにする．

2. 割り込み禁止

SR（ステータスレジスタ）をスタックに保存し，SR=0x2700 を設定して走行レベルを 7 に変更することで，割り込みを禁止する．これにより，割り込み処理の最中に別の割り込みが発生して処理が競合するのを防ぐ．

3. p_body() または v_body() の呼び出し

D0 レジスタの値を確認し，以下のように処理を分岐させる：

・D0 == 0 の場合，D1（セマフォ ID）を引数として p_body(D1)を実行する。

これは P 操作（セマフォ取得）を行い，セマフォが使用可能でない場合はタスクを休眠状態にする可能性がある．

・D0 == 1 の場合，D1 を引数として v_body(D1)を実行する．

これは V 操作（セマフォ解放）を行い，待機しているタスクがあれば実行可能状態に戻す．

4. 割り込み禁止の解除

SR をスタックから復元し，元の割り込みレベルに戻す．これにより，通常の割り込み処理が再開され，システムの割り込み管理が正常に動作する．

5. レジスタの復元

割り込み処理開始時にスタックに保存したレジスタを復元し，元のタスクの状態を回復する．

6. 割り込み処理の終了

RTE（Return from Exception）命令を実行し，割り込み処理を終了する．これにより，処理が中断されたタスクまたは新たにスケジュールされたタスクが再開される．

test2.c

このプログラムファイルには，セマフォを用いたタスクの同期を検証するためのテストプログラムが記述されている．これは，テキスト 1.4.5 節の trap #0 のシステムコール番号 5 の機能追加実装及び 2.5.6 節の低難度 Option: skipmt() の実装を踏まえて，タスクの同期

処理を確認できるように実装したものである．そのため，まずどのようにタスクの同期を実現したかについて説明を行う．

セマフォ応用

タスクの同期処理

本実装では，複数のタスク(task_k(), task_0()…)をセマフォを用いて同期させる仕組みを構築する．ここでは，各タスク(task_k(), 1 ≤ k ≤ N)の終了順は順不同だが，全タスクが停止した後に一斉に再開できるように実装を行う．

同期の仕組み

1. 共有変数とセマフォの準備
   - nttask（終了したタスク数）を共有変数として定義.
   - semaphore[0]（カウンティングセマフォ）で nttask の更新を保護.
   - semaphore[1]（バイナリセマフォ）でタスクの待機・再開を制御.
2. タスクの動作
   - 各タスク(task_k())は作業終了後，クリティカルセクション(P(0), V(0))で nttask を更新し, P(1)で待機状態に入る．
3. 監視タスク（task_0()）の役割
   - nttask == N（全タスクが停止）を検出すると, nttask = 0 にリセットし, V(1)を N 回実行して全タスクを再開する．
   - skipmt()を実行し, CPU 資源を節約.

この仕組みにより，各タスクの終了順に関係なく，全タスクの再開を統一的に管理することができる．

この test2.c 内では， main()でカーネルの初期化，セマフォの設定，タスクの登録を行い，スケジューラ（begin_sch())を開始する．タスクの実行中，セマフォを用いてタスク間の同期を制御し，全タスクが停止した後に一斉に再開できるようにしている．

以下に，スケジューラに登録しているタスクの詳細を示す.

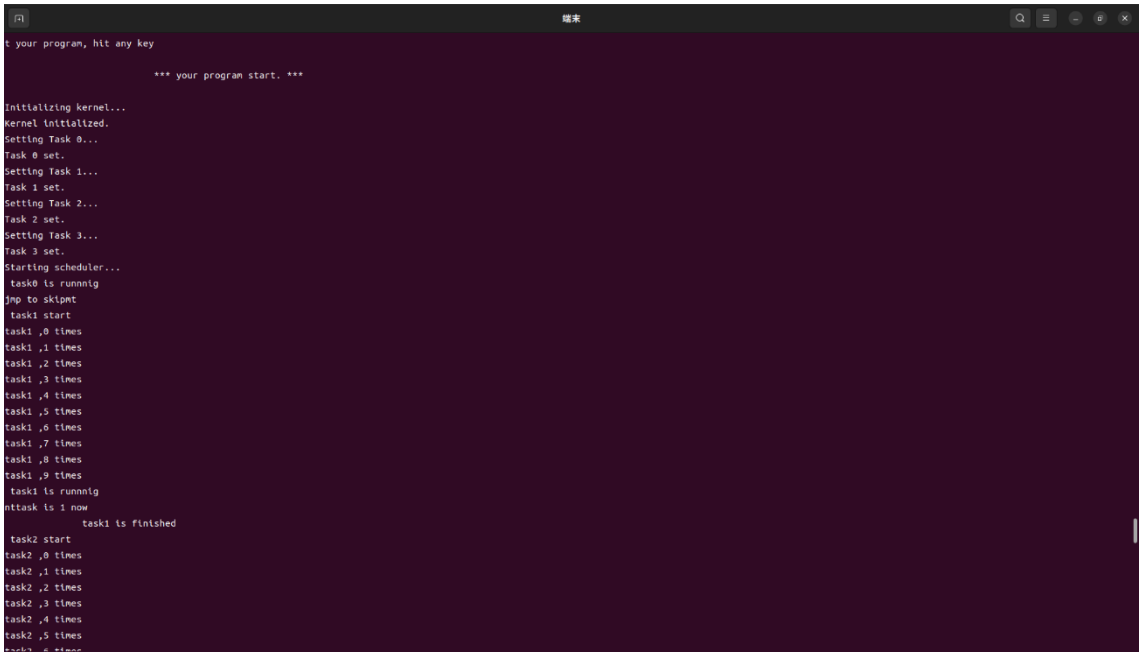
task_0（監視タスク）

他のタスク(task1, task2, task3)の進行を監視し，全タスクが停止(nttask == N)したら，再開（V(1)）を行う，タスクの同期処理後, skipmt()を実行し，計算資源を解放する．

task1, task2, task3（作業タスク）

各タスクは異なるループ（K, 2K, 3K 回)を実行するように記述されており，処理の負荷がそれぞれ異なる値に設定されている．各ループ終了後, P(0)でクリティカルセクションを確保し，nttask をインクリメント(nttask++)し，実行終了を通知するような動作をする．そしてすぐに V(0)でクリティカルセクションを解放し，P(1)で待機状態に入る．全タスクが終了するまでそれぞれ待機し，task_0 によって V(1)が呼ばれると，再び動作を開始する．

最後に，実際にこのタスクを実行した際の結果を示す．



これを見ると，処理の負荷の異なる 3 つのタスクがタイマ割り込みによって一定時間ずつ実行されているが，処理が進むにつれ負荷の軽い処理から終了し，残りのタスクのみが切り替えられ，すべてのタスクが終了した際に，監視タスクが働き，同期した後再開されており，上に示した同期処理が実現できていることがわかる．

5. プログラムにおいて注意したこと

1. スタックとタスクコントロールブロックの理解

テーマ 2 のマルチタスクカーネル作成や実際にタスクを用意してテストを行っていく中

で，スタックとその情報の管理に関する理解は不可欠であり，テーマ 2 においてもっとも時間をかけた作業であった．特に，タスクコントロールブロックの持つ情報などをスタックに積んでいく際のアドレス計算や，積まれ方の仕様などは図を用いて念入りに情報を共有して，班員全員が動作を理解して実装できるように注意した．

2. セマフォに関する理解

マルチタスキングを実装するなかで，タスクの同期処理を実現するために，本実験ではバイナリセマフォとカウンティングセマフォを使い分ける必要があった．もちろんセマフォの操作をするための関数を正確に作成するのはもちろんのこと，クリティカルセクションなど，タスクがどのようにセマフォを利用していくのかを構造建てて考える必要があり，これも班員で時間をかけて話し合い理解を行った．

3. ヘッダファイルの整理

我々の班では，作業を分野ごとに振り分け，各々が違う関数を作成するような方法を取った．そのため，各関数の引数や変数の定義などについての一貫性を保つためにヘッダファイルで適切にそして整理して宣言し，矛盾ができるだけ発生しないように注意した．

4. クラウドを利用した情報の共有

本実験では，各々に PC が割り当てられ，それにボードを接続してデバッグやテストを行う必要があったため，最新の実行環境を班員同士で常に共有しておかなければ，誰かが解決した問題で他の人が詰まってしまうような事態が発生する危険があった．我々の班では，全員が Git hub のようなプラットフォームを使いこなせる訳ではなかったため，Onedrive の共有フォルダを通して，修正・改善したコードの共有を行い，スムーズに実装が進むように注意した．

6. プログラムに発生した問題とその原因，解決方法

addq()、removeq() の有効活用

我々の班では，作業を「マルチタスク機能」「タイマ」「セマフォ」の 3 つに分け，それぞれの担当チームが開発を進めた．その結果，基本的に同じ担当部門内のメンバーのみで意見を交わしながらコードを作成することになり，他の担当部門との連携が少なくなってしまった．そのため，各チームが作成したコードを統合してテストを行った際，異なるチームが定義した変数や関数のやり取りがうまくいかず，プログラムが正しく動作しなかった．

特に，タスクの実行・休眠を制御する sleep()・wakeup() 関数が機能せず，コードを調査したところ，タスクの状態を適切に更新する addq()・removeq() を使用せずに，複数の変

数を用いて手動でタスクの登録や new_task の更新を行っていたことが原因であると判明した．この実装では，タスク管理の整合性が取れず，エラーが発生していた．

この問題を解決するために，自分を含めたマルチタスク機能を担当するメンバーを交えて話し合い，コードを大幅に整理・削減し，addq()・removeq() を適切に活用する形に修正した．その結果，タスク管理の効率が向上し，プログラムの動作が安定し，テストも正常に動作するようになった．

以下に，修正を行う前後での sleep()・wakeup() のコードを示す．

・修正前の sleep()・wakeup()

```
int sleep(int a){
    /* a := セマフォ ID*/
    int b = semaphore[a].task_list; /* b := セマフォ[a]の先頭タスク ID*/
    if(b == NULLTASKID){
        semaphore[a].task_list = curr_task; /* セマフォ[a]が空のとき現在のタスクをセマフォの先頭へ */
        task_tab[curr_task].next = NULLTASKID;
    }else{
        int c = task_tab[b].next;
        while(c != NULLTASKID){ /* セマフォキューの末尾を辿り、たどり着いた時、 */
            b = c; /* b= 末尾のタスクの ID */
            c = task_tab[b].next; /* c= NULLTASKID */
        }
        task_tab[b].next = curr_task; /* セマフォキューの末尾に現在のタスクを登録 */
        task_tab[curr_task].next = NULLTASKID;
    }

    sched();
    swtch();
}

int wakeup(int a){
    /* a := セマフォ ID */
    int b = semaphore[a].task_list; /* b:= セマフォ[a]の先頭タスク ID */
    if(b != NULLTASKID){
```

```
        int c = task_tab[b].next; /* c := セマフォ[a]の二番目のタスク ID */
        task_tab[b].next = NULLTASKID;

        semaphore[a].task_list = c; /* セマフォ[a]の先頭のタスク ID を c に設定 */

        if(ready == NULLTASKID){
            ready = b; /* ready キューが空のとき先頭のタスク ID を b にする */
        }else{
            int d = task_tab[ready].next;
            while(d != NULLTASKID){ /* ready キューの末尾をたどり、たどり着いた時、 */
                c = d; /* c= 末尾のタスクの ID */
                d = task_tab[c].next; /* d= NULLTASKID */
            }
            task_tab[c].next = b; /* ready キューの末尾に b を登録 */
        }
    }
}
```

・修正後の sleep()・wakeup()

```
int sleep(int a)
{ /* a := セマフォ ID*/
  addq(&(semaphore[a].task_list), curr_task);
  sched();
  swtch();
}

int wakeup(int a)
{
  TASK_ID_TYPE task_id = removeq(&(semaphore[a].task_list));
  addq(&ready, task_id);
}
```

7. 考察

テーマ 2 を通して,TCB やセマフォ，タイマ割り込みを用いてマルチタスクカーネルを作成した．今回は，タスクの切り替えを一定の周期でのタイマ割り込みをもとに行い，タスク

の順番は FIFO で決定されることから, Round-Robin スケジューリング方式をとっていると言える.

その中でも, セマフォは重要な役割を果たしていたといえる. 特に, 共有リソースに対する排他制御を実現する P 操作と V 操作を活用することで, リソース競合を防ぎつつ, タスク間の同期を確実に行うことができた。一方で, セマフォを適切に管理しない場合にはデッドロックのリスクがあることも確認できた.

また, TCB を用いたタスク管理では, タスクの状態遷移を明確に管理でき, タスク切換えの際には, 現在のタスクの情報を保存し, 次のタスクの情報を復元するプロセスを円滑に実現できた. これにより, タスク切り替えを効率よく実現しており, 今回のマルチタスキングの根幹を支えていると実感する結果となった.

さらに, Round-Robin スケジューリング方式では, タイマの割り込み周期がシステムに大きな影響を与えていると考えられる. 具体的には, 間隔を短く設定すると切り替えが頻発しオーバーヘッドが増加する一方, 長く設定しすぎるとタスクの応答性が低下すると考えられる. そのため, これ以降の実験では, このトレードオフを考慮し, システムに最適な時間を選定することが重要であると考えられる.

このように, 今回の実験を通じて, セマフォや TCB を活用した Round-Robin スケジューリング方式のマルチタスキングは, 効率性と公平性を両立する優れた手法であるといえる. またそれと同時に, リソース管理やタスク設計において多くの工夫が求められることも事実であるため, より複雑なマルチタスクシステムの設計や応用にむけてこの学びを活かしていきたい.

テーマ 3：応用

1. テーマ 3 についての説明

テーマ 3 では, テーマ 2 で作成したマルチタスクカーネルを拡張し, 2 つの RS232C ポートを用いた I/O 処理を並列実行可能な環境を構築する. これにより, 2 つのポートを用いたデータ通信やマルチユーザー環境のシミュレーションが可能となる.

具体的には以下を実装・拡張する.

1. I/O 関数の改造
   inbyte()や outbyte()をそれぞれのポートに対応させ, 独立した I/O 処理が可能になる

よう修正する． UART1 を RS232C ポート 0, UART2 を RS232C ポート 1 に割り当てる．

2. ファイルディスクリプタによるポート指定
   read()や write()の第 1 引数であるファイルディスクリプタ（fd）を拡張し，どのポートで I/O を行うかを指定可能にする．

3. fdopen()の実装
   各ポートに対応するストリームを適切に割り当てることで，複数のポートを扱う I/O 操作を効率化する．

2. テーマ 3 の実験全体における位置づけ

テーマ 3 の目的は以下の 2 点に集約される．

1. マルチタスクの機能拡張
   複数のポートを同時に制御することで，より実践的なマルチタスク環境を構築する．

2. 排他制御の応用
   複数のタスクが同時に I/O リソースを使用する場合，セマフォを用いて適切に排他制御を行い，データ競合を防ぐ．

また，このテーマの後には 2 ポートを活用したアプリケーションの開発も行う．例えば, 2 人用の対戦ゲームや双方向の通信システムなどが考えられる．これにより，マルチタスク処理の実装だけでなく，実際のシステム開発における応用力も養う．

3. プログラムのリスト（新規作成部）

・inchrw.s

```
.global inbyte
.include "equdefs.inc"

.text
.even

inbyte:
```

```
    link.w  %a6, #-4
    movem.l %d1-%d3/%a0, -(%sp)

inbyte_loop:
    move.l  #SYSCALL_NUM_GETSTRING, %d0
    move.l  %sp, %a0
    adda.l  #28, %a0
    move.l  (%a0),%d1

    move.l  %a6, %d2
    sub.l   #1,  %d2
    move.l  #1, %d3
    trap    #0

    cmpi.l  #0, %d0
    beq     inbyte_loop
    move.b  'A', LED0

    move.b  '0', LED1

    move.b  -1(%a6), %d0

    movem.l (%sp)+, %d1-%d3/%a0
    unlk    %a6
    rts
```

・**outchr.s**

```
.global outbyte
.include "equdefs.inc"

.text
.even

outbyte:
    movem.l %d1-%d3/%a0, -(%sp)
```

```
outbyte_loop:
    move.l  #SYSCALL_NUM_PUTSTRING, %d0

    movea.l %sp, %a0
    adda.l  #20, %a0
    move.l (%a0), %d1

    move.l  %sp, %d2
    addi.l  #27, %d2

    move.l  #1, %d3
    trap    #0

    cmpi.l   #0, %d0
    beq outbyte_loop

    movem.l  (%sp)+, %d1-%d3/%a0
    rts
```

・csys68k.c

```
#include <stdio.h>
#include "mtk_c.h" // マルチタスクカーネル用ヘッダー
#include <stdarg.h>
#include <fcntl.h>
#include <errno.h>

extern void outbyte(int ch, unsigned char c);
extern char inbyte(int ch);

int read(int fd, char *buf, int nbytes)
{
  char c;
  int i;
  int ch;

  if (fd != 0 && fd != 3 && fd != 4)
```

```
{
  return EBADF;
}
else if (fd == 0 || fd == 3)
{
  ch = 0;
}
else
{
  ch = 1;
}

for (i = 0; i < nbytes; i++)
{
  c = inbyte(ch);

  if (c == '¥r' || c == '¥n')
  { /* CR -> CRLF */
    outbyte(ch, '¥r');
    outbyte(ch, '¥n');
    *(buf + i) = '¥n';

    /* } else if (c == '¥x8'){ */ /* backspace ¥x8 */
  }
  else if (c == '¥x7f')
  { /* backspace ¥x8 -> ¥x7f (by terminal config.) */
    if (i > 0)
    {
      outbyte(ch, '¥x8'); /* bs  */
      outbyte(ch, ' ');   /* spc */
      outbyte(ch, '¥x8'); /* bs  */
      i--;
    }
    i--;
    continue;
  }
```

```
    else
    {
      outbyte(ch, c);
      *(buf + i) = c;
    }

    if (*(buf + i) == '¥n')
    {
      return (i + 1);
    }
  }
  return (i);
}

int write(int fd, char *buf, int nbytes)
{
  int i, j;
  int ch;

  if (fd != 1 && fd != 2 && fd != 3 && fd != 4)
  {
    return EBADF;
  }
  else if (fd == 1 || fd == 2 || fd == 3)
  {
    ch = 0;
  }
  else
  {
    ch = 1;
  }

  for (i = 0; i < nbytes; i++)
  {
    if (*(buf + i) == '¥n')
    {
```

```
      outbyte(ch, '¥r'); /* LF -> CRLF */
    }
    outbyte(ch, *(buf + i));
    for (j = 0; j < 300; j++)
      ;
  }
  return (nbytes);
}

int fcntl(int fd, int cmd, ...)
{
  if (cmd == F_GETFL)
  {
    return O_RDWR;
  }
  else
  {
    return 0;
  }
}
```

・**test3.c 内 mapping 関数**

```
//---------------------mapping 関数-------------------
void mapping()
{
    com0in  = fdopen(3, "r");
    com0out = fdopen(3, "w");
    com1in  = fdopen(4, "r");
    com1out = fdopen(4, "w");
}
```

4. プログラムの説明

・**inchrw.s**

テーマ 3 では，複数ポートからの入出力を受け付けるために，read()と write()を呼び出す際に，第 1 引数である fd を用いて，ポートの指定を行う．そのため，その内部で呼び出され

る inbyte に関しても fd に対応した形に変更する必要があり, fd をもとにシステムコール trap #0 で指定されたポートを利用する形に変更を行った.

具体的には, %a0 にスタックポインタの値を保存し, そこに退避させたスタックや戻り先の PC の分の領域を考慮したオフセットを加算し, 第 1 引数が格納されている場所から確実に値をとりだし, %d1 格納するという命令を記述している.

・outchr.s

outbyte も上記の理由から, 同様に fd に対応した形へと変更を行った.

具体的にはここでも, %a0 にスタックポインタの値を保存し, そこに退避させたスタックや戻り先の PC の分の領域を考慮したオフセットを加算し, 第 1 引数 fd が格納されている場所から確実に値をとりだし, %d1 格納するという命令を記述している. そして, このままシステムコールにチャンネルとして渡されるという流れになっている.

また, 新たに%a0 を退避させているために, 実際のデータの格納領域を示すためのオフセットも変更されている.

・csys68k.c

read の変更点

ここでは資料に基づき, 読み書き可能性を考慮したデバイスマッピング判定をもとにポートを選択できるように, 変更を行っている.

具体的には, fd が 0 または 3 の場合はシリアルポート 0 を, fd が 4 の場合はシリアルポート 1 を使用し, その他の fd に対しては EBADF（無効なファイルディスクリプタエラー）を返すように実装している. また, ポート選択のために新たに ch という変数を定義し, inbyte に引数として渡す値として利用している.

write の変更点

write も同様に, 資料に基づいたデバイスマッピング判定を導入し, fd が 1, 2, 3 の場合はシリアルポート 0 を, fd が 4 の場合はシリアルポート 1 を使用し, その他の fd では EBADF を返すようにしている. またここでも, ポート選択のために新たに ch という変数を定義し, outbyte に引数として渡す値として利用している.

fcntl の新規実装

fcntl()は関数引数の個数が可変であるため, 最初に stdarg.h をインクルードしたうえで, cmd が F_GETFL の場合に, O_RDWR(入出力可能)を返し, その他には 0 を返すように実

装をしている．これは，本来ならば fd に対する open()時にのフラグをカーネルに問い合わせる必要があるが，実験カーネルでは管理していないためである．

・test3.c 内 mapping 関数

この関数では，入出力ストリームをファイルディスクリプタに割り当てるための C ライブラリ関数である fdopen()を用いて，FILE 構造体へのポインタとしての大域変数を準備するための関数である．fdopen()は，FILE *fdopen(int fd, const char *mode); と定義されており，fd はファイルディスクリプタ，mode は fd に対する入出力モードを示しており，読み込みの場合は”r”，書き込みの場合は”w”を指定する．ここでは，RS232C に対応するものとして，

com0in：　UART1 からの読み込み
com0out：　UART1 への書き込み
com1in：　UART2 からの読み込み
com1out：　UART2 への書き込み

と準備している．

5. プログラムにおいて注意したこと

プログラムで注意した点は，inbyte・outbyte を変更する際に，read()・write()の第 1 引数である fd の値を取って来る際のスタックの扱いである．資料にあるように引数は，スタックに技から左へと積まれ，さらに char(8 ビット)，short(16 ビット)のデータは符号拡張命令(EXT)を用いて 32 ビットデータに符号拡張したのちスタックに積まれるといった処理が施されるためそれを考慮したオフセット計算が必要であった．そのため，班員で図を書きながら確認をし合って値を決めて，確実に値を取ってこられるようにした．

また，mapping()や fcntl()をどこに記述し，extern 宣言を行えばいいのかも注意して確認し，プログラムが問題なく動作するように記述した．

6. プログラムに発生した問題とその原因，解決方法

上記に示した通り，引数の処理のされ方には様々なルールがあったが，はじめはこれを理解していなかったため，2 ポートを使った入出力が全くできていなかった．そして改めて資料を読み込んでいくとこのルールの存在に気づき，修正することができた．修正の途中でも様々な値を試して，そのたびに LED を使ったデバッグでどこまで実行できているか，またどんな引数が渡されているかのテストをしながら確認することで，正確なオフセットを求めることができた．

7. 考察

テーマ 3 では, inbyte, outbyte, read(), write()などの改良, mapping(), fcntl()の作成を通して, 2 ポートで入出力を行う事ができるようになった. これにより複数の画面を使用してタスクを実行していくことができるようになり, テーマ 2 で実現したマルチタスク性を十分に発揮するプログラムを動かすことが可能になったと言える.

個人課題

1. 個人課題の概要

個人課題では, 簡易的なトランプの High&Low ゲームの実装を行った. 実際に, 実装した内容は以下の通りである.

- 1~13 までのカードが, シャッフルされて山札として設定される.
- はじめの 1 枚をめくり数字を確認する
- 2 人のプレイヤーは山札の次の数字が先程のカードより大きいか小さいかを予想し, それぞれ High であれば 1, Low であれば 0 を入力する
- 山札をめくり, 予想があたったものにポイントが加算される.
- 指定したラウンドまで繰り返し, ポイントが多いほうが勝者となる.

この課題内での, マルチタスク性を活かした部分は, プレイヤーが回答をする部分で, まず両者の入力を同時に受け付けたうえで, そろってラウンドを重ねていくために, 両者ともに解凍が終わるまで, 結果判定・及び次のラウンドに進まないようになっている. また, 乱数を生成するタスクを独立に用意し, はじめの手札をシャッフルする部分で活かしている.

2. プログラムのリスト

・test3.h

```
#ifndef TEST3_H
#define TEST3_H

// カード枚数（1〜13）
#define TOTAL_CARDS 13

//「数字カード」だけを持つシンプルな構造体
```

```
typedef struct
{
    int number; // 1〜13
} CARD;

// グローバルな山札の配列（13 枚）
extern CARD cards[TOTAL_CARDS];

// カードの初期化（1〜13 を設定）
void init_card();

// カードのシャッフル
void shuffle_card();

#endif // TEST3_H
```

・**test3.c**

```
#include <stdio.h>
#include "mtk_c.h"
#include "test3.h"

CARD cards[TOTAL_CARDS];

// -------------------- グローバル変数 --------------------
int round_count;
int top_index;

int score1;
int score2;
int guess1;
int guess2;

int is_finish;
int winner;
// -------------------- 乱数生成 --------------------
volatile unsigned int rand_counter = 2;
```

```
void random_generator()
{
    while (1)
    {
        rand_counter = (rand_counter * 1103515245 + 12345) & 0x7FFFFFFF;
    }
}

unsigned int get_random()
{
    return rand_counter;
}

// -------------------- カード初期化 --------------------
void init_card()
{
    for (int i = 0; i < TOTAL_CARDS; i++)
    {
        cards[i].number = i + 1;
    }
}

void shuffle_card()
{
    for (int i = TOTAL_CARDS - 1; i > 0; i--)
    {
        unsigned int r = get_random() % (i + 1);
        int temp = cards[i].number;
        cards[i].number = cards[r].number;
        cards[r].number = temp;
    }
}

// -------------------- ラウンド表示 --------------------
void first_display_round()
```

```
{
        fprintf(com0out, "----------------------¥n");
        fprintf(com0out, "Round %d¥n", round_count);
        fprintf(com0out, "Current Number: %d¥n", cards[top_index].number);
        fprintf(com0out, "Player0 Score=%d, Player1 Score=%d¥n", score1, score2);

        fprintf(com1out, "----------------------¥n");
        fprintf(com1out, "Round %d¥n", round_count);
        fprintf(com1out, "Current Number: %d¥n", cards[top_index].number);
        fprintf(com1out, "Player0 Score=%d, Player1 Score=%d¥n", score1, score2);
}

// -------------------- Player 0 --------------------
void player0(){
    P(0);
    while (1)
    {
        fprintf(com0out, "Player 0: 次のカードは High(1) か Low(0) か? (Score=%d): ",
score1);

        int buf1;
        fscanf(com0in, "%d", &buf1);

    if (buf1 == 1){ //High
        guess1 = 1;}
    else if (buf1 == 0){    //Low
        guess1= -1;}
    else{
        guess1 = 1;     //default High
        }

        V(0);
        V(2);
        P(3);
    }
}
```

```
// -------------------- Player 1 --------------------
void player1(){
    P(1);
    while (1)
    {
        fprintf(com1out, "Player 1: 次のカードは High(1) か Low(0) か? (Score=%d): ",
score2);

        int buf2;
        fscanf(com1in, "%d", &buf2);

    if (buf2 == 1){ //High
        guess2 = 1;}
    else if (buf2 == 0){    //Low
        guess2= -1;}
    else{
        guess2 = 1;     //default High
        }

        V(1);
        V(2);
        P(3);
    }
}

// -------------------- 結果判定 --------------------
void judgment()
{
    while (1)
    {
        P(0);
        P(1);
        P(2);
        P(2);
        int next_number = cards[++top_index].number;
```

```
if (next_number > cards[top_index - 1].number)
{
    if (guess1== 1)
        score1++;
    if (guess2 == 1)
        score2++;
}
else if (next_number < cards[top_index - 1].number)
{
    if (guess1 == -1)
        score1++;
    if (guess2 == -1)
        score2++;
}

round_count++;

fprintf(com0out, "----------------------¥n");
fprintf(com0out, "Round %d¥n", round_count);
fprintf(com0out, "Current Number: %d¥n", cards[top_index].number);
fprintf(com0out, "Player0 Score=%d, Player1 Score=%d¥n", score1, score2);

fprintf(com1out, "----------------------¥n");
fprintf(com1out, "Round %d¥n", round_count);
fprintf(com1out, "Current Number: %d¥n", cards[top_index].number);
fprintf(com1out, "Player0 Score=%d, Player1 Score=%d¥n", score1, score2);

if (round_count >= 5 || top_index >= TOTAL_CARDS)
{
    is_finish = 1;
    winner = (score1 > score2)   ? 0
             : (score1 < score2) ? 1
                                 : -1;

    fprintf(com0out, "*** Game Over! ***¥nWinner: %s¥n",
            (winner == -1)  ? "Draw"
```

```
                    : (winner == 0) ? "Player 0"
                                    : "Player 1");

            fprintf(com1out, "*** Game Over! ***¥nWinner: %s¥n",
                    (winner == -1)  ? "Draw"
                    : (winner == 0) ? "Player 0"
                                    : "Player 1");
            while(1){
            }
        }
        V(3);
        V(3);
        V(0);
        V(1);
    }
}


//---------------------mapping 関数-------------------
void mapping()
{
    com0in  = fdopen(3, "r");
    com0out = fdopen(3, "w");
    com1in  = fdopen(4, "r");
    com1out = fdopen(4, "w");
}

// -------------------- main 関数 --------------------
int main()
{
    mapping();
    init_kernel();
    init_card();
    shuffle_card();
    first_display_round();

    semaphore[0].count = 1;
```

```
    semaphore[1].count = 1;
    semaphore[2].count = 0;
    semaphore[3].count = 0;

    set_task((char*)random_generator);
    set_task((char*)player0);
    set_task((char*)player1);
    set_task((char*)judgment);

    begin_sch();
    return 0;
}
```

3. プログラムの説明

・グローバル変数

ゲームの状態を管理

int round_count：ラウンド数

int top_index：現在のカードの位置

プレイヤーの管理

int score1, score2：プレイヤーのスコア

int guess1, guess2：プレイヤーの予想(High:1/Low:0)を管理

フラグと勝者

int is_finish：終了フラグ

int winner：勝者の情報

・random_generator()

この関数は，擬似乱数を生成するための関数である．無限ループ内で計算を繰り返し，グローバル変数 rand_counter に擬似乱数を生成して格納している．乱数を生成する仕組みについては，線形合同法（Linear Congruential Generator, LCG）と呼ばれる乱数生成アルゴリズムを用いており，無限ループ内で乱数を生成し続けるため、常に最新の乱数を取得できる仕組みになっている．しかしこのプログラム内では，最初のシャッフルの際に 1 回呼び出すだけとなっている．

・get_random()

この関数は，グローバル変数 rand_counter に格納されている最新の擬似乱数を取得して返す関数である．別タスクとして動作する random_generator 関数によって生成された乱数を利用するためのインターフェースを提供している．

・init_card()

この関数は，カードゲームに使用するカードのデッキを初期化するための関数である．for ループを使用して，cards 配列内の各要素に 1 から順に番号を順番に割り当る用になっている．

・shuffle_card()

この関数は，カードデッキ（cards 配列）をランダムに並び替えるためのシャッフル機能を提供するものである．init_card 関数で順番に初期化されたカードを，乱数を利用してランダムな順序に並び替えることで，ゲーム用のデッキを作成します．詳細な動作は以下の通りである．

1. デッキの末尾からシャッフル

   逆順ループ（for 文）を使用して，配列の末尾から順番にシャッフルを行う．現在のインデックス i より小さい範囲でランダムな位置 r を選択し，i 番目のカードと r 番目のカードを交換する．

2. ランダムなインデックスの生成

   関数 get_random()を使用して乱数を取得し，それを i+1 で割った余り（% (i + 1)）を計算して，0 から i までのランダムなインデックス r を生成する．

3. カードの入れ替え

   cards[i].number と cards[r].number の値を交換することで，シャッフルを実現する．

このシャッフルの方法は，Fisher-Yates 法に基づいており，配列のサイズ（TOTAL_CARDS）に応じて動的に動作するため，カードの枚数が変更されても問題なく対応可能な仕組みになっている．

・first_display_round()

この関数は，ゲームの各ラウンド開始時に現在の状況をプレイヤーに表示するための機能を提供している．ラウンド数，現在のカードの番号，各プレイヤーのスコアをそれぞれのシリアルポート（com0out と com1out）を通じて出力する．

・player0(), player1()

これらの関数は，それぞれ Player 0 と Player 1 の入力処理を行うタスクとして設計されている．プレイヤーが「次のカードの値が前のカードより高い（High: 1）か、低い（Low: 0）か」を予想し、その結果を変数 guess1 または guess2 に記録する．また，セマフォを使用してタスク間の同期を行い，ゲームの進行を制御する．

・mapping()

省略

・main()

この関数は，カードゲームの初期設定とタスクの登録を行い，タスクスケジューラを起動する役割を担っている．全体的な初期化処理と，ゲームを開始するための準備を行う．経じめに，セマフォ 2 のカウントを 0 にすることで，判定タスクが待機状態になるようにしている．

・タスク間の流れとセマフォ操作

以下に, 1 ラウンドでのタスク間の流れとセマフォ操作の順序を示す.

1. Player 0 の入力
    - P(0)を実行してセマフォの待機を解除し, Player 0 が入力を開始.
    - 入力完了後：
        - V(0)：Player 0 の入力完了を通知.
        - V(2)：判定タスクを実行可能にする．

2. Player 1 の入力
    - P(1)を実行してセマフォの待機を解除し, Player 1 が入力を開始.
    - 入力完了後：
        - V(1)：Player 1 の入力完了を通知.
        - V(2)：判定タスクを実行可能にする．

3. 判定処理
    - 判定タスクは P(0), P(1)、P(2)（2 回）を実行して，両プレイヤーの入力が完了していることを確認.
    - 次のカードを判定し，スコアを更新.
    - ゲーム進行状況を出力.
    - 次のラウンドの準備として, V(3)を 2 回呼び出して，両プレイヤーが次のラウン

ドで入力を再開可能にする．また，V(0)と V(1)を呼び出して，次のラウンドで各プレイヤータスクが入力を開始可能にする．

4. 個人課題まとめ

個人課題を通じて，期待どおりに動作しない場面が多く，原因を探ることに多くの時間を費やした．そのため，ゲームのルールを変更したり，機能を削減することになり，マルチタスク性を最大限に発揮した制作物にはならなかった．しかし，セマフォを用いたマルチタスクの仕組みやその重要性について理解を深めることができた点は，大きな収穫だと感じている．

今後，さらに機能を追加したり，より本格的なトランプゲームに近づけた実装を行う際には，優先度ベースのタスクスケジューリングや非同期 I/O の導入が非常に有用だと考えている．これらを実現することで，効率的で拡張性の高いマルチタスクシステムが構築できると考えられ，今後の課題として引き続き取り組みたい．